DSA Assignment Documentation  
By: Gabriel Muller

Contents

[1.0 Preamble 1](#_Toc528572987)

[1.1 Discussion of Abstract Data Types (ADTs) 1](#_Toc528572988)

[2.0 Setup 2](#_Toc528572989)

[2.1 Classes 2](#_Toc528572990)

[3.0 Part 1 Major Decisions 2](#_Toc528572991)

[3.1 Part 1 Code Discussion 2](#_Toc528572992)

[3.1.1 Part 1 Filter Functions 3](#_Toc528572993)

[3.2 Part 1 Conclusion/Future Choices 3](#_Toc528572994)

[4.0 Part 2 Major Decisions 3](#_Toc528572995)

[4.1 Part 2 Code Discussion 4](#_Toc528572996)

[4.2 Part 2 Conclusion/Future Choices 4](#_Toc528572997)

[5.0 Part 3 Major Decisions 4](#_Toc528572998)

[5.1 Part 3 Code Discussion 4](#_Toc528572999)

[5.2 Part 3 Conclusion/Future Choices 5](#_Toc528573000)

[5.2.1 Part 3 Dictionaries/Hash Tables 5](#_Toc528573001)

[5.2.2 Part 3 Dictionaries/Hash table Conclusion 6](#_Toc528573002)

[6.0 Part 4 Major Decisions 6](#_Toc528573003)

[6.1 Part 4 Code Discussion 6](#_Toc528573004)

[6.2 Part 4 Conclusion/Future Choices 6](#_Toc528573005)

[7.0 Test Harness 7](#_Toc528573006)

[8.0 Conclusions 7](#_Toc528573007)

[9.0 References 7](#_Toc528573008)

# 1.0 Preamble

To preface any of the documentation or any decisions made, all the code was implemented through Python rather than Java, this decision simply came from a background of undertaking Fundamentals of Programming (FOP) which is taught in Python rather than Object Oriented Program Design (OOPD) which is taught in Java. The assignment uses implementation of various ADTs and methods which were introduced and implemented weeks prior to the assignment with updates being made to optimize them and make them more effective. Also some path location for files may need to be changed as the file may not be in the same location for the operating user.

# Discussion of Abstract Data Types (ADTs)

There were two main types of ADTs used that were usedthese two being linked lists and graphs. Linked lists are the main type of ADT and are used essentially for everything in Parts 1,2 and 3 of the assignment. Linked lists are a way of storing data that allow for nodes to be inserted in an order and has a head (start) and tail (end) nodes, each node has a pointer to the next node and the list can be traversed from the beginning to the very end. Linked lists are extremely useful and easy to implement into most situations and it’s the reason why they were used so much. The other ADT that was used is graphs, and this was used for Part 4. Graphs allow for vertices to be created with links to other vertices and weights, based on the weight of these links, a shortest path can be calculated to get from point A to point B in the shortest time possible. This is useful and implemented in Part 4 to create a suitable itinerary based on the results of Part 3. The other ADT that was attempted to be implemented was Hash Tables, this would allow for key, value combinations and allows for the summation of said key value combinations. This is useful as it essentially acts as a dictionary and would have been able to make Part 3 be fully functional. Other ADTs such as trees could have been used for things such as storing information, sorting or key/value pairs but I was unsure as to how to implement them for those. The code for Queues is implemented through Linked Lists rather than arrays but an array may have been more useful for faster processing speed.

# 2.0 Setup

Prior to a discussion of the code itself, I will firstly be talking in regards to the major decisions made throughout the assignment. These are what I believe to have been major decisions and will be broken down according to each part of the assignment. Part 1 consists of filtering and ordering data, Part 2 consists of being able to search for candidates of a list based on a substring, Part 3 dealt with marginal seats of a party and lastly Part 4 was concerned with creating an itinerary of locations to visit based on the results from Part 3.

# 2.1 Classes

In terms of classes, there are 4 total classes with each representing a section. The reason for this is that most parts were relatively straight forward to implement and didn’t require a lot of inheritance to make the code neater/more effective. Personally a few more classes would have been suitable in my case considering the amount of functions that were introduced in each part of the assignment as well as the repetition. Each of these is further discussed in each Conclusion/Future choices for each part. There are also the classes for each ADT, the Linked Lists class comes from Practical 4 along with stacks/queues, and the graphs ADT comes from Practical 6 which also inherits code from stacks/queues found in Practical 4. These classes will not be discussed in detail, but the reasoning for using each ADT will be discussed in each section

# 3.0 Part 1 Major Decisions

In Part 1 the two initial decisions that were made were, those being to use Linked Lists as the main ADT type and bubble sort algorithm for sorting. Linked Lists were used simply due to the fact that they are easy to unpack and easy to store data into, all results from the filtering were stored as Linked Lists of a larger LinkedList, this means that essentially every results inside the LinkedList is its own little list holding all the candidate information, this makes it extremely easy to unpack results separately and show individual fields if necessary.

The other major decision was to use bubble sort. The reason behind this is quite simple, it is the easiest of the sorting methods to implement (just adaptation for looking at a list of lists rather than a singular list) and no extra marks appeared to be awarded for optimization of a sorting algorithm, hence time complexity did not really appear to matter in this section.

# 3.1 Part 1 Code Discussion

Part 1 ended up having a lot of functions that realistically should’ve been shortened or put into different classes due to all the repetition present. The starting function is used to basically separate the three main filter fields, states, parties and divisions into separate Linked Lists which can then be used to compare the values in each to the according user inputs. The function Option1 is the starting function that selects the first filter by option, from this it is then passed onto to Option2 in which the user can decide if they would like to sort by more than one option. If the user only wants to filter by one option it passes them through to the corresponding functions otherwise the program keeps going to pass it through to different functions.

If the user does select to filter by another field or multiple other fields it passes the user through to other functions otherwise it asks the user what states, parties or divisions they would like to see the results for. Each function past this is an adaptation of the original stateS, partyS and divS but just adding another field to filter by, so there is a function for states and then party, state and then division, party and then state, etc. This is quite ineffective and not the best implementation, however it does allow for better readability of the code and its easier to overall find and alter things separately if needed. I do wish to change this to essentially have a function for state, party and division, and according to the user input return those in their correct order (instead of having a separate function for each, the user would select say party and state, and it would first return the party function and then the state function). If this were to be implemented, it would easily reduce the overall amount of functions. The filter functions allow for multiple inputs, the reasoning behind this is hypothetically let’s say the user wants to filter by state, party and division, the results would be extremely limited if they were only able to enter 1 of each and hence extending that to multiple inputs allows for more results and wider range of results. The user input for amount of each filter display is not properly checked, hence the user can enter up to infinity stats, parties or divisions.

The sortBy checks what field the user would like to sort the results by (options being surname, state, party and division). Depending on user selection it returns a value which corresponds to the column where that field is found, this then passes it on to the bubbleSort function.

Lastly the the miscellaneous functions include showUResul, showResult and CSVexp. showResult and showUResult are self-explanatory, it just displays the sorted results in their sorted order or unsorted results. CSVexp was done as a bit of an extra regarding whether the user would like to export the results to a csv file.

# 3.2 Part 1 Conclusion/Future Choices

In conclusion implementation of methods and functions for Part 1 are done in a somewhat suboptimal way. There are multiple ways to improve it, the 2 biggest/main improvements would be through cutting down functions (could easily reduce the code by ~300+ lines I would imagine) and implementing a quicker sorting algorithm than bubble sort. Another way to improve would be to select the next filter option one at a time and creating an inner menu inside this part 1, this means essentially if the user selects to filter by state, a menu would then popup asking if they would also like to filter by state/party or if they would like to proceed to saving the results/displaying. This would make it a more interactive user environment and enhance experience whilst cutting down the computing overhead of having so many functions. Overall Part 1 works accordingly but just not implemented very efficiently. Also limiting the user value input to the length of each of the unique linked list.

This part should have been implemented with other classes, each relating to the functions that could be called accordingly (Eg: a class for state, a class for party and a class for division) this would have allowed for a reduction in repetition. The reason that it wasn’t implemented is the time constraint, unfortunately I did not fully grasp the concepts of classes and inheritance until recently and going through and changing the code to be more effective would have taken a long time, this is a fault on my behalf. A class should have also been introduced as a general exit function that could be applied to each part, instead it is implemented as function for the same reasons.

# 4.0 Part 2 Major Decisions

Part 2 consisted of searching a candidate based on a substring of their name. Much like Part 1 for similar reasons, Linked Lists were used therefore I won’t discuss them in the same detail, but essentially Linked Lists were used to store in unique values for state, parties and surnames. Originally this section was setup to find the corresponding user string in ANY part of the surname, through essentially using   
if userinput in ‘abcde’:   
 print(corresponding row) (eg: if the user input was an, it would include someone with the last name Andrews, but also someone with the last name Van Halen).

The biggest decision was the use of the .find() method, this method could be used to compare the surname of a candidate to that of the userinput in the first character of the string. This allowed for finding only those candidates that had the user input at the beginning of their name rather than anywhere in their name. I believe that this method is allowed since it is built into python and it isn’t any real type of ADT, it’s only used as a comparison tool of the user input to every row, much like checking for if userinput in.

# 4.1 Part 2 Code Discussion

The first function acts as a way to split candidate names, states and parties into only the unique attributes and store them into linked lists which can later be used for comparison. The candSearch function is the beginning function for this, it asks the user what the substring they would like to search for is and if they would like to filter by state and/or party. Similarly to Part 1 there is a function for every choice, this is not very optimal, it does have increased readability and understanding but it creates too much redundancy and repetition that is unwanted, these functions are each names accordingly, candStateParty, candState, candParty, etc. Each function stores the result which is then passed through to showResults and/or CSVexp, one of them is used to show the results and CSV exp is used to export the results in a csv format. There was also an issue with checking whether the user matched any of the starting string names , hence the program will run regardless of user input and hence the program won’t display any outputs or be able to export anything to the CSV.

# 4.2 Part 2 Conclusion/Future Choices

This section of the assignment is relatively short and straight forward in comparison to the rest. The best ways to improve it would be to once again reduce the number of overall functions, limiting it to candidate, state and party and calling each one accordingly, implement a way to filter by multiple substrings, states and parties. Once again this part has a very high computational overhead that could quite easily be reduced.

Similarly to Part 1, a class could have been introduced for every filter option and should have been implemented accordingly. Due to time limitations and not fully understanding classes this was not done.

# 5.0 Part 3 Major Decisions

Part 3 required to filter marginal seats based on a default of 6% or whatever the user input was as well as filtering by a specific party. Once again Linked Lists were used to store unique values (parties in this case). The biggest decision made in this section was the use of the original file that was made for this section. This function does not operate the exact way it needs to in order to comply to Part 4, this was essentially my original take on what was required before realizing it wasn’t really useful for Part 4. Inside the assignment folder there is another folder (named NOTWORKING) which contains the other two attempts at Part 3, one which contains dictionaries which are banned, and the other which uses Hash Tables, but it was not working, these two are discussed at the end. Therefore, realistically the biggest decision was the dilemma in relations to whether to use the file which uses dictionaries and is fully functional and in compliance to the standards of Part 4 or use the original file which does not realistically fulfil any of the requirements of Part 4. Due to the potential loss of 20 marks for the use of dictionaries I decided to use the original file.

The original file basically works by allowing the user to select a division (if they would like) and after that select parties to compare the votes to in that division. What should have been done is calculating the votes of each party to every other party in the same division and checking if any of them are within a 6% margin of each other (or user input). This obviously doesn’t correlate to Part 4 as it outputs only a single result.

# 5.1 Part 3 Code Discussion

The first function is concerned with opening all the csvs and merging them into a single file (given that the file does not yet exist in the directory) and save the file as margin.py. To clarify this file was not used in the final version this is due to the party Shooters, Fishers and Farmers that contained a comma and once splitting up the function via comma it would throw an error. Instead I manually altered the file and named it test.csv which is the file that is used, I attempted using .replace() to replace all instances but could not get it working in time for submission.

Following that and saving the unique division names into a separate LinkedList. The split function then does a couple things. Firstly we set the user value to be None, this in regards to the 6% margin which the user can choose to change, the reason we initiate it as None is because later on for margin comparison the loops check whether the user entered a value or not, if they did it uses the new value as a comparison otherwise it overwrites the user value of None to be the newly set user value. The function also initializes 4 separate counters to account for votes according to the functions and linked to store all the votes from the chosen division/party. Once again, a function is created for every separate user choice, based on whether they want just a division, just a single party, two party comparison, etc.

Each one then does a comparison of value results and pass it through to check whether the margin was changed by the user and then compares whether the result are within the margin or not. The results are then passed through to CSVexp which checks if the user would like to export the results, this CSV function is useless in this case as the file that is exported cannot be used for anything and doesn’t store anything more than a value (if the value is in the marginal threshold range). The extra function that was written is named exit, this function just ensures that according to the user input it always passes them through to a check if they would like to restart this part or return to the main menu.

# 5.2 Part 3 Conclusion/Future Choices

In conclusion Part 3 was unfortunately not submitted according to what was necessary. The reason for leaving this part in is to allow there to be something implemented for this section that theoretically works at producing a margin and to also be able to discuss the future use of Hash tables. Overall this part is hard to discuss as it does not meet the standards however a lot of discussion can be done on how it should have been implemented properly with hash tables (which is the discussion following from this section).

The reasoning behind a single class for this one is relatively straight forward, there isn’t a lot of functions that are repeated in this section in comparison to others. This allows for the use of a single class and inheritance would be seemingly meaningless and would only potentially introduce more problems into the code.

# 5.2.1 Part 3 Dictionaries/Hash Tables

Much like the original file the dictionaries (hash table) file would split up into unique divisions and would then initiate unique linked lists for both the chosen party and all other parties. Each list stored a tuple containing state, division, party and the votes. Two separate dictionaries were also initiated which are used following this section, based on the key/value results, if they were in the dictionary, the dictionary would add them up based on they key, if they weren’t in the dictionary a key/value pair would be created. Then for each key (division) that matched it would do a comparison of votes for each party (based on user input) to one another in that division. If the results were within the margin, the state and division would be appended to a list (as a tuple). The user could then choose to display the results, write the results to a csv or exit the program. The reason why the state and division are saved into the linked list is just to ensure that in case a division name was present in multiple states, the state could be used as a comparison for Part 4.

The implementation with hash tables didn’t work due to the get method. The problem here that I was unable to fix was that it would only acquire the first value containing that key, although multiple values existed holding that key and hence this did not allow for multiple values in the same key. The easiest way I could imagine to fix this would be to use chaining along the hash table, the hash table is only implemented (poorly) with linear probing, this is suboptimal and creates clusters which are in no way useful.

# 5.2.2 Part 3 Dictionaries/Hash table Conclusion

Part 3 is definitely the biggest discussion point of the assignment due to the use of lists. Essentially that is the biggest thing to be changed in a future implementation, removing lists and replacing them with Hash Tables. Dictionaries would be the most optimal way of implementation however due to the unit’s structure and specifications it isn’t allowed to be used. The others things to discuss would be the use of tuples and subtuples, I believe this to be the best method for extracting and storing the data, the reason for this is due to divisions having multiple parties within them, and a division potentially being present in two different states, although the key is quite long it ensures that matching keys are able to be found easily and there isn’t any repetition of keys throughout. The real problem and the reason hash tables weren’t used it simply due to their poor implementation on my behalf.

# 6.0 Part 4 Major Decisions

The biggest decision in this section is obviously the use of Graphs and the implementation of Dijkstra’s algorithm as the traversal algorithm. The implementation of Graphs was done as the itinerary is based on locations and times between locations, graphs allow for each location-destination to be created as a vertex-link connection (eg: Perth to Curtin is created as a link). Each vertex has multiple locations attached to it and graphs make this implementation relatively easy.

The second big decisions would have been the use of Dijkstra’s algorithm. There are two reasons for the implementation of this over any other algorithm. Firstly it get the job done, it creates a minimum spanning tree to allow each location to be visited only once and find the smallest possible connection to each one through a while loop comparison. The second reason is that it’s one of the easiest methods to implement, although it is not the most optimal or the fastest algorithm it is relatively easy to implement when compared to other algorithms (Floyd-Warshall/Kruskal). Therefore Dijkstra’s algorithm would have been the best/easiest choice for this section.

As this section contained three separate files, the relevance and use of each of the files is further discussed in Section 6.1.

# 6.1 Part 4 Code Discussion

The initializing function of Part 4 loads in all the files that are necessary for use in this part. The start function opens all the files necessary, the division file is the file created from Part 3, every division from that file is made into a vertex and added to a linked list. Following from this the airports distance file is then looked at and each airport is made into its own vertex and added to the same list of locations as before. The interesting thing to note here is that the airports distance file has values for both planes and cars, this can be useful in the traversal but will be talked about more in-depth later. Then based on the linked list created which contains both divisions from Part 3 and Airports from the file that was read in, if any of those locations match any location from the electorate distance csv file (both as a starting and end point), an edge is created between the locations.

Past this point there isn’t a lot left in the program to do, this is because Djikstra’s algorithm was not implemented. It gives the user the ability to display the edge list and then carry out a Breadth First Search based on the starting location the user would like. There is an extra exit function that just occurs after everything to allow the user to either restart the process or send them back to the main menu.

For some unknown reason I am unable to return the visited list in BFS and print it out, hence the repetition of the final BFS lists and for this unknown reason I am also unable to get anything to work part BFS hence the whole program ends when the user tries to carry out BFS.

# 6.2 Part 4 Conclusion/Future Choices

There isn’t a lot to discuss in this section due to Part 3 not doing its job properly. This means that the file used for the itinerary is the same every time regardless of anything else, the only part that can change is the BFS according to user input (even then it doesn’t change very much). The choice for BFS is due to me being unable to get DFS working and apart from that there isn’t much discussion in relation to the actual code. In the future it would also be nice to get BFS fully implemented and running properly.

Much like Part 3, Part 4 does not contain a lot of code, this is because most of the code is found inside the graphs code which inherits from other pieces of code. This allows for this section to be relatively short programming wise and hence no need for external classes. The only function that should be its own class is the exit() function, as mentioned in Part 1, this was essentially a general function I could have used throughout my code but chose not to due to time limitations and not fully understanding classes/inheritance.

In regards to future implementation there is quite a bit. Inside the folder NOTWORKING there is code that contains another attempt at Part 4, in this code the implementation of edges with weights works, however BFS stops working properly and hence I decided to not use this file, inside there the graphs file also includes an attempt to implement Djikstra’s algorithm to no real avail. In reality the algorithm and weight of the edges would be found inside the Part 4 file and Part 3 would have also been properly implemented alongside it

# 7.0 Test Harness

In regard to designing a test harness, this is a bit complicated. As Python students we were never really introduced to test harnesses in Python, in this case using try/except method is a bit hard to implement as we haven’t been exposed to it enough, the only real place where it should have been implemented is in Part 1 when checking for the user input of integers for how many states/parties/divisions they would like to display. Apart from this I’m unsure how to create a test harness outside of physically testing my code as I go along. This is the reason why a test harness was not created.

# 8.0 Conclusions

Overall the implementation of all the code was suboptimal at best. Everything has quite a large overhead and due to the many unnecessary functions created a lot of the code could have been shortened and made a lot less repetitive. Part 1 could have worked with a simple function for each of the three filter choices rather than a function for every option. Part 2 could have worked better in the same way as Part 1, but also allowing the user to search for more than one substring of characters. Part 3 could have been better implemented in a lot of ways, most of this is discussed in the conclusion of Part 3 but the biggest takeaway is the implementation of hash tables to allow for proper implementation that links to Part 4. Part 4 is missing a lot of what it needs to have, specifically a search algorithm, due to complications weights were also unable to be added to edges however an external Part 4 file found in the NOTWORKING folder does have that part of the code working. Overall for future reference there is a lot of better methods of implementations and things that could have been done much better without a time constraint being present.

# 9.0 References

Harry Walters, Provided me with code for \_\_getitem\_\_ and \_\_setitem\_\_ for linked lists which allowed them to be indexed removing the need to use a python inbuilt list for the sorting function in Part 1

Peter Mortesen, if s.find(“is”) == -1:  
<https://stackoverflow.com/questions/3437059/does-python-have-a-string-contains-substring-method>

“7.1 string – Common string operations”. 24th of October 2018. Docs Python.  
<https://docs.python.org/2.7/library/string.html#string.find>